An Empirical Study of the Control and Data Planes (or Control Plane Determinism is Key for Replay Debugging Datacenter Applications)

Gautam Altekar
Ion Stoica

An Empirical Study of the Control and Data Planes (or Control Plane Determinism is Key for Replay Debugging Datacenter Applications)[*(2446)*]

Gautam Altekar
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

Replay debugging systems enable the reproduction and debugging of non-deterministic failures in production application runs. However, no existing replay system is suitable for datacenter applications like Cassandra, Hadoop, and Hypertable. For these large scale, distributed, and data intensive programs, existing methods either incur excessive production overheads or don't scale to multi-node, terabyte-scale processing.

In this position paper, we hypothesize and empirically verify that *control plane determinism* is the key to record-efficient and high-fidelity replay of datacenter applications. The key idea behind control plane determinism is that debugging does not always require a precise replica of the original datacenter run. Instead, it often suffices to produce some run that exhibits the original behavior of the *control-plane*–the application code responsible for controlling and managing data flow through a datacenter system.

1 Introduction

The past decade has seen the rise of large scale, distributed, data-intensive applications such as HDFS/GFS [15], HBase/Bigtable [9], and Hadoop/MapReduce [10]. These applications run on thousands of nodes, spread across multiple datacenters, and process terabytes of data per day. Companies like Facebook, Google, and Yahoo! already use these systems to process their massive data-sets. But an ever-growing user population and the ensuing need for new and more scalable services means that novel applications will continue to be built.

Unfortunately, debugging is hard, and we believe that this difficulty has impeded the development of existing and new large scale distributed applications. A key obstacle is non-deterministic failures–hard-to-reproduce program misbehaviors that are immune to traditional cyclic-debugging techniques. These failures often manifest only in production runs and may take weeks to fully diagnose, hence draining the resources that could otherwise be devoted to developing novel features and services [21]. Thus *effective tools for debugging non-deterministic failures in production datacenter systems are sorely needed.*

Replay-debugging technology (a.k.a, deterministic replay) is a promising method for debugging non-deterministic failures in production datacenters. Briefly, a replay-debugger works by first capturing data from non-deterministic data sources such as the keyboard and network, and then substituting the captured data into subsequent re-executions of the same program. These replay runs may then be analyzed using conventional tracing tools (e.g., GDB and DTrace [8]) or more sophisticated automated analyses (e.g., race and memory-leak detection [*REF*], global predicates [13, 17], and causality tracing [12]).

1.1 Requirements

Many replay debugging systems have been built over the years and experience indicates that they are invaluable in reasoning about non-deterministic failures [4, 7, 11, 13, 14, 16, 17, 19, 20, 23]. However, no existing system meets the unique demands of the datacenter environment.

Always-On Operation. The system must be on at all times during production so that arbitrary segments of production runs may be replay-debugged at a later time.

Unfortunately, existing replay systems such as liblog [14], VMWare [4], PRES [20] and ReSpec [16] require all program inputs to be logged, hence incurring high throughput losses and storage costs on multicore, terabyte-quantity processing.

Whole-System Replay. The system should be able to replay-debug *all nodes* in the distributed system, if desired, after a failure is observed.

Providing whole-system replay-debugging is challenging because datacenter nodes are often inaccessible at the time a user wants to initiate a replay session. Node failures, network partitions, and unforeseen maintenance are usually to blame, but without the recorded information on those nodes, existing systems may not be able to provide replay.

High Replay Fidelity. No replay debugging system can be considered useful if it cannot reproduce the underlying errors leading to program failures.

Unfortunately, existing replay systems such as ODR [5] (our prior work), ESD [23], and SherLog [22] support efficient datacenter recording, but they make no guarantees of the fidelity of reproduced runs – they may or may not exhibit the underlying error that was originally observed.

1.2 Contributions and Non-Contributions

The contributions of this work are two fold.

A Hypothesis. First, we put forth the hypothesis that *control plane determinism* is sufficient and necessary for debugging datacenter applications. The key observation behind control-plane determinism is that, for debugging, we don't need a precise replica of the original production run. Instead, it often suffices to produce some run that exhibits the original run's *control-plane* behavior. The control-plane of a datacenter system is the code responsible for managing or controlling the flow of data through a distributed system. An example is the code for locating and placing blocks in a distributed file system.

Supporting Evidence. Second, we back up the above hypothesis with experimental evidence. In particular, we show that, for datacenter applications, (1) the control plane is considerably more prone to bugs that the data-plane and (2) the data plane rather than the control plane is responsible for almost all I/O consumed and generated. Taken together, these results suggest that by relaxing the determinism guarantees to control-plane determinism, a datacenter replay system can meet all of the aforementioned requirements.

While our goal is to advocate control plane determinism, we do not discuss the mechanism for achieving and implementing it in a real replay system. We defer these details to the `DCR` system [6].

2 Overview

We present the central hypothesis of this work and then describe the criteria that must be met to verify it.

2.1 Hypothesis: The Control Plane is Key

We hypothesize that, for debugging datacenter applications, a replay system need *not* produce a precise replica of the original run. Rather, it generally suffices for it to produce *some* run that exhibits the original *control-plane* behavior.

The control-plane of a datacenter application is the code that manages or controls data-flow. Examples of control-plane operations include locating a particular block in a distributed filesystem, maintaining replica consistency in a meta-data server, or updating routing table entries in a software router. The control plane is widely thought to be the most error-prone component of datacenter systems. But at the same time, it is thought to consume only a tiny fraction of total application I/O.

A corollary hypothesis is that datacenter debugging rarely requires reproducing the same *data-plane* behavior. The data-plane of a datacenter application is the code that processes the data. Examples include code that computes the checksum of an HDFS filesystem block or code that searches for a string as part of a MapReduce job. The data plane is widely thought to be the least error-prone component of a datacenter system. At the same time, experience indicates that it is responsible for a majority of datacenter traffic.

2.2 Testing Criteria

To show that our hypothesis holds, we must empirically demonstrate two widely held but previously unproven assumptions about the control and data planes.

Error Rates. First, we must show that control plane rather than the data plane is *by far* the most error prone component of datacenter systems. If the control plane is the most error prone, then a control-plane deterministic replay system will be able to reproduce a majority of bugs. If the control plane is not, then such a replay system will not be able to meet the "reproduce most bugs" requirement, and thus will be of limited use in the datacenter.

Data Rates. Second, we must show that the data plane rather than the control plane is *by far* the most data intensive component of datacenter systems. If so, then a control plane deterministic replay system is likely to incur negligible record mode overheads – after all, a control plane deterministic system makes no attempt to record and replay the data plane. If, however, this does not hold, then even control plane determinism is likely to be too strong for the datacenter, and our hypothesis will be falsified.

3 Classification

To verify our hypothesis, we must first classify program code as belonging to either the control or data planes. Achieving a perfect classification, however, is challenging because the notions of control and data planes are tied to program semantics, and hence call for considerable developer effort and insight to distinguish between the two types of code. Consequently, any attempt to manually classify every source line of large and complex systems is likely to provide unreliable and irreproducible results.

Rather than expend considerable manual effort, we employ a more reliable, semi-automated classification method. This method a operates in two phases. In the first phase, we manually identify and annotate user data. By user data we mean any data inputted to the datacenter application with semantics clear to the user and opaque to the system. In the second phase, we automatically identify the static program instructions that operate on (i.e., are tainted by) the annotated user data. Any instructions tainted by data are classified as belonging to the data plane; other instructions are classified as part of the control plane.

3.1 Annotating User Data

To annotate user data we employ two source code annotation functions: `MARK_CHANNEL` and `MARK_MEMORY`.

`MARK_CHANNEL(fd)` take as input a file descriptor for a file or socket. It's most useful when all data transfered on a particular file or socket channel is known to be user data. For example, suppose we want to denote that all data a Hypertable client reads from a database input file should be marked as user data. Then we may do so as follows:

```
LoadDataSource::LoadDataSource(
      std::string fname, ...) {
   m_fin.open(fname.c_str());
   MARK_CHANNEL(m_fin.get_fd());
}
```

In the above example, we mark the data file being loaded after it is opened. The annotation tells our classification runtime that data returned by all subsequent transactions (i.e., reads and writes) on the file descriptor should be treater as user data.

`MARK_MEMORY(addr, len)` takes as input a memory region. It's most useful when channel transactions may contain either control or data plane information. For example, if updates received by Hypertable Range server are to the global table directory (METADATA table), then those updates are part of the control plane–they operate on an internal, non-user table. But if updates are to a user-created table, then they are part of the data plane. The `MARK_MEMORY` annotation allows us to make the distinction as follows:

```
void RangeServer::update(
      TableIdentifierT *table,
      BufferT &buffer) {

   const bool is_root =
      strcmp(table->name,
            "METADATA") == 0;

   /* Data plane only if not a METADATA
      update. */
   if (!is_root) {
      MARK_MEMORY(buffer.buf,
            buffer.len);
   }

   ...
```

In the above code, we mark the incoming data as user data only if the target table is not the root (i.e., METADATA) table.

3.2 Tracking User Data Flow

Approach. We employ a simple dynamic taint flow analysis to track user data. We chose a dynamic rather than a static approach for two reasons. First, we wanted accurate results. Unfortunately, a static approach is prone to false positives and/or negatives due to pointer aliasing issues. Second, we wanted to track user data though the entire application, including dynamically linked libraries and any dynamically generated code. Many applications employ both for control and data plane processing, and ignoring them would likely pollute our results. Unfortunately, both techniques have traditionally been difficult for static analysis.

Of course, the main drawback of a dynamic taint flow analysis is that it only gives you results for one execution. This is problematic because if an instruction isn't executed in one run, then we have no way to classify it. Also, it's possible that even if an instruction operated on tainted data in one run, it may not in another run. We deal with these issues by performing the taint on multiple executions and on a varied set of inputs. Furthermore, we consider the instruction as part of the data plane only if it is consistently tainted across executions.

Implementation. Our taint flow mechanism operates at the instruction level with the aid of the Valgrind binary translation framework, in a manner most similar to that of the Catchconv project [18]. Briefly, Valgrind translates x86 into a minimalistic, RISC-like intermediate language. We perform our analysis on this intermediate language. The analysis tracks data at byte granularity. We use a simple hash-table to remember which bytes are and are not tainted by user data.

4 Evaluation

Here we evaluate the testing criteria for our hypothesis (see Section 2.2) on real datacenter applications. In short, we found that both clauses of the testing criteria held true. In particular, we found that control plane code is the most error prone (with an average 98% of all bugs belonging to it), and that data plane code is the most data intensive (accounting for as much as 99% of all application I/O).

4.1 Setup

Applications. We test our hypothesis on two real-world datacenter applications: *Cloudstore* [1] and *Hyptertable* [2].

Cloudstore is a distributed filesystem written in 40K lines of multithreaded C/C++ code. It consists of 3 sub-programs: the master server, slave server, and the client. The master program consists mostly of control-plane code: it maintains a mapping from files to locations and responds to file lookup requests from clients. The slaves and clients have some control-plane code, but mostly engage in control plane activities: the slaves store and serve the contents of the files to and from clients.

Hypertable is a distributed database written in 40K lines of multithreaded C/C++ code. It consists of 4 key sub-programs: the master server, metadata server, slave server, and client. The master and metadata servers are largely control-plane in nature–they coordinate the placement and distribution of database tables. The slaves store and serve the contents of tables placed there by clients, often without the involvement of the master or the metadata server. The slaves and clients are thus largely data-plane entities.

Workloads and Testbed. We chose the workloads to mimic datacenter operation. Specifically, for *Hypertable*, 2 clients performed concurrent lookups and deletions to a 10 GB table of web data. Hypertable was configured to use 1 master server, 1 meta-data server, and 1 slave server. For *Cloudstore*, we made 2 clients concurrently get and put 10 GB gigabyte files. We used 1 master server and 1 slave server.

All applications were run on a 10 node cluster connected via Gigabit Ethernet. Each machine in our cluster operates at 2.0GHz and has 4GB of RAM. The OS used was Debian 5 with a 32-bit 2.6.29 Linux kernel. The kernel was patched to support `DCR`'s interpositioning hooks. Our experimental procedure consisted of a warmup run followed by 6 trials, of which we report only the average. The standard deviation of the trials was within three percent.

	Code Size (Instructions)		
Application	Control (%)	Data (%)	Total
CloudStore			
Master	100	0	120K
Slave	99	1	160K
Client	99	1	120K
Hypertable			
Master	100	0	110K
Metadata	100	0	150K
Slave	99	1	180K
Client	99	1	130K

Figure 1: Plane code size for each application component. As expected, the control plane accounts for almost all of the code in an application.

4.2 Error Rates

Metrics. We gauge errors rates with two metrics. The first metric, *plane code size*, is the number of instructions in the control or data plane of an application. It indirectly measures the complexity and hence potential for developer mistakes of a plane's code. The second metric, termed *plane bug count*, is the number of bug reports encountered in each component over the system development lifetime.

Method. Measuring plane size is straightforward. We simply looked at the results of our classification analysis (see Section 3). Measuring plane bug count was more challenge because it required inspecting and understanding all defects in the application's bug report database. For each defect, we isolated the relevant code and then used our understanding of the report and our code classification to determine if it was a control or data plane issue.

	Reported Bugs		
Application	Control (%)	Data (%)	Total
Hypertable			
Master	100	0	18
Metadata	100	0	12
Slave	94	6	32
Client	-	-	0

Figure 2: Reported bug count for each application component classified into control and data plane bins. As predicted, the control plane has the most bugs. CloudStore numbers are not given because it does not appear to have a bug report database.

4.2.1 Plane Code Size

Figure 1 gives the size in static instructions–an indicator of program complexity–for the control and data planes. At the high level, it conveys two points.

First, some application components such as the Hypertable Master and Metadata are entirely composed of control plane code. No measurements were needed for them as these components were entirely control plane by design–no user visible data flows through them.

Second, components such as the Hypertable Slave and Client have some data plane code, but are still largely control plane in nature. On the Hypertable client, for example, the small amount of data plane code is responsible for loading and parsing user data. On the Hypertable slave, a small amount of data plane code is needed to parse incoming updates and hash row-keys into an internal key-value store.

4.2.2 Plane Bug Count

Figure 2 classifies application bugs into control and data planes. At the high level it shows that most datacenter applications bugs originate in the control plane.

To determine why the control plane was so error-prone, we inspected the code to identify trends that would explain the high error rate. The inspection revealed two reasons for the error-prone nature of the control plane.

First, the control plane bugs tends to be complex–an artifact of the need to efficiently control the flow of large amounts of data. For example, Hypertable migrates portions of database from node to node in order to achieve an even data distribution. But the need to do so introduced Hypertable issue 63 [3] — data corruption error that happens when clients concurrently access a migrating table.

By contrast, the data plane tends to be simple and leverages previously developed code bases (e.g., libraries). Most often, it performs a serial operation on a bag of bytes without extensive coordination with other components. For example, a large part of a Hypertable's data plane is devoted to compressing and decompressing a Hypertable cell-store. But this is a relatively simple operation in that it involves invoking a well-test compression library routine.

	I/O Bytes Consumed/Generated		
Application	Control (%)	Data (%)	Total
CloudStore			
Master	100	0	128 MB
Slave	1	99	20.6 GB
Client	1	99	10.6 GB
Hypertable			
Master	100	0	96 MB
Metadata	100	0	128 MB
Slave	1	99	20.5 GB
Client	1	99	10.2 GB

Figure 3: Input/output (I/O) rates of all application components, each broken down by control and data planes. For components with high data rates, almost all I/O is generated and consumed by the data plane.

4.3 Data Rates

Metric. We present the control and data plane input/output (I/O) rates of each system component. Data is considered input if the plane reads the data from a communication channel. Data is considered output if the plane writes the data to a communication channel. By communication channel, we mean reads and writes to file descriptors (e.g., those connect to the tty, a file, a socket, or to a device).

Method. To measure the I/O rates, we interposed on common inter-node communication channels. To do so, we intercepted read and write system calls. If the data being read/written was influenced/tainted by user data, then we considered it data plane I/O plane; otherwise it was treated as control plane I/O.

Results. Figure 3 gives the data rates for the control and data planes. At the high level, the results shows that the data plane is by far the most data intensive component. More specifically, the control plane code accounts for at most 1% of total application I/O in components that have a mix of control and data plane code (e.g., Hypertable Slave and Client). Moreover, in components that are exclusively control plane (e.g., the Hypertable Master), the overall I/O rate is orders of magnitude smaller than those that have data plane code.

5 Conclusion

A replay debugger for datacenter applications must reproduce distributed system errors and provide lightweight recording. In this paper, we've argued that a datacenter replay system can do both by shooting for control plane determinism–the idea that is suffices to produce some run that exhibits the original run's control plane behavior. To support our argument, we provide experimental evidence showing that the control plane is responsible for most errors and that it operates at low data rates. Taken together, these results support our belief that control plane determinism can enable practical datacenter replay.

References

[1] Cloudstore. `http://kosmosfs.sourceforge.net/`.

[2] Hypertable. `http://www.hypertable.org/`.

[3] Hypertable issue 63. http://code.google.com/p/hypertable/issues/.

[4] Vmware vsphere 4 fault tolerance: Architecture and performance, 2009.

[5] ALTEKAR, G., AND STOICA, I. Odr: output-deterministic replay for multicore debugging. In *SOSP* (2009).

[6] ALTEKAR, G., AND STOICA, I. Dcr: Replay debugging for the data center. Tech. Rep. UCB/EECS-2009-108, EECS Department, University of California, Berkeley, May 2010.

[7] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE* (2006).

[8] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *USENIX* (2004).

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[10] DEAN, J., AND GHEMAWAT, S. Mapreduce: a flexible data processing tool. *CACM 53*, 1 (2010).

[11] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *VEE* (2008).

[12] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[13] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[14] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX* (2006).

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003).

[16] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Online multiprocessor replay via speculation and external determinism. In *ASPLOS* (2010).

[17] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *NSDI* (2007).

[18] MOLNAR, D. A., AND WAGNER, D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Tech. Rep. UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[19] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS* (2009).

[20] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP* (2009).

[21] VOGELS, W. Keynote address. CCA, 2008.

[22] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[23] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *EuroSys* (2010).